爱创课堂前端培训

JS 基础

第 6 天课堂笔记（本课程共 8 天）

班级：北京前端训练营 7 期

日期：2017 年 5 月 17 日

爱创课堂官网 ：www.icketang.com

目录

复习

作用域：

变量的作用域：JS 没有块级用做域，只有函数可以管住作用域。

局部变量：当函数执行才定义，函数执行完毕之后销毁。所以在作用域外的任何地方都不能访问咱们的局部变量。

全局变量：可以在任何地方访问的到。用作：信号量，累加的作用。

函数的作用域：只有在它作用域内才可以被执行。

```
function outer(){
            var a = 1;
            function inner(){
```

```
            console.log(a);
        }
        inner();
    }
    outer();
    inner();
```

闭包：函数的闭包天生存在，但是我们看不见，只有通过某种方法才可以在外面看见闭包的效果。函数可以记住执行时外部环境和内部语言。

```
function outer(){
        var a = 1;
        function inner(){
            console.log(a);
        }
        return inner;
    }

    var i = outer();
    i();
```

数组：array，类型是引用类型。直接用数组字面量定义。[]。

索引值（index）。index 的最大值 length - 1。

```
arr[index]
```

数组的属性 length。

```
arr.length
```

数组的首尾操作：

```
shift()      //删除第一位
unshift()    //在首位增加
pop()        //删除最后一项
push()       //在末尾增加
```

concat()。数组的拼接，连接。

slice(start,end)。数组的拆分。表示截取 start 到 end 之间的数，包括 start 不包括 end。参数可以是正数可以是负数，end 还可以不写。

splice(index，howmany,elements)；

删除：splice(3,4);

插入:splice(3,0,elements);

替换：splice(3,4,elements);

join（）表示吧数组转换为字符串。

reverse()。倒序

sort()。第一种没有参数，就表示简单排序。按照字符编码顺序。

第二种有参数，

```
// 升序
    sort(compareFunction(a,b){
        if(a > b){
```

```
                return 1;
            }else if(a == b){
                return 0;
            }else{
                return -2;
            }
        })
```

一、字符串的属性和方法

1.1 字符串的属性

字符串的属性和数组一样也有 length 属性。表示字符串的长短。每一个字母，汉字，数字，标点符号，空格都算一个长度。直接使用点方法。

```
var str = "今天天气太闷了，感觉气温有 35°C";
console.log(str.length);
```

```
17
```

字符串也有索引值 index。也是从 0 开始。

1.2 字符串的方法

charAt()：表示索引值对应的字符。

```
var str = "今天天气太闷了，感觉气温有 35°C";
// charAt()参数是索引值，返回值是该索引值对应的字符。
console.log(str.charAt(3));
```

```
气
```

```
// 字符串的遍历
for(var i = 0 ; i <= str.length - 1 ; i ++){
    console.log(str.charAt(i));
}
```

indexOf()：表示该字符首次出现的位置。如果该字符串存在返回的是字符串所在的索引值，字符串不存在返回的是-1。

```
// indexOf()表示该字符首次出现的位置。
var str = "今天天气太闷了，感觉气温有35°C";
console.log(str.indexOf("天"));
```

```
1
```

concat()：表示字符串的拼接。参数可以是一个字符串或者多个，用逗号隔开。

```
//方法表示字符串的拼接,返回一个新的字符串，源字符串不变。
var str = "今天天气太闷了，感觉气温有35°C";
var str2 = str.concat("我需要空调","某同学想吃冰棍");
console.log(str);
console.log(str2);

```

```
今天天气太闷了，感觉气温有35°C
今天天气太闷了，感觉气温有35°C我需要空调某同学想吃冰棍
```

slice(start,end)方法表示截取索引值从 start 到 end 之间的字符串。

包括 start 不包括 end。

可以用正值，可以用负值，还可以不写 end。

```
// slice()表示截取一段区间的字符串
var str = "今天天气太闷了，感觉气温有35°C";
var str2 = str.slice(3,9);
var str3 = str.slice(-9,-3);
var str4 = str.slice(3);
```

```
今天天气太闷了，感觉气温有35°C
气太闷了，感
感觉气温有3
气太闷了，感觉气温有35°C
```

split()方法表示用某个字符截取字符串得到一个新数组。

参数：字符

```
// split()用字符截取得带新的数组
var str = "aaabccccbddjbjjkbhhjj";
var arr = str.split("b");
console.log(arr);
```

```
["aaa", "cccc", "ddj", "jjk", "hhjj"]
```

substr(start，howmany)方法：表示从某个索引值开始截取一段数目的字符串。

参数 start 表示开始截取的索引值

howmany 表示要截取的个数。

```
var str = "今天天气太闷了，感觉气温有 35°C";
console.log(str.substr(2,5));
console.log(str.slice(2,7));
```

```
天气太闷了
天气太闷了
```

substring()：表示截取一段字符串从 start 开始到 end 不包含 end。

不可以用负值。

```
// substring()方法
var str = "今天天气太闷了，感觉气温有 35°C";
console.log(str.substring(2,7));
// substring()方法可以书写从大到小，会自行判断两个数值的大小，包括小的索引值，不包括大的索引值。
console.log(str.substring(7,2));
```

toUpperCase()方法表示转换为大写字母。

toLowerCase()方法表示转换为小写字母。

```
var str = "qfjfkjkka";
var str2 = str.toUpperCase();
var str3 = str2.toLowerCase();
console.log(str2);
console.log(str3);
```

```
QFJFKJKKA
qfjfkjkka
```

将“everything is good in its season”，转为每个单词的首字母大写。

即“Everything Is Good In Its Season”

```
var str = "everything is good in its season";
// 将每个单词的首字母大写。
// 先得到每一个单词
var arr = str.split(" ");
// 每个单词的首字母大写
for(var i = 0 ; i <= arr.length - 1 ; i ++){
    arr[i] = arr[i].charAt(0).toUpperCase() + arr[i].slice(1);
}
// 数组转换为字符串
var str1 = arr.join(" ");
console.log(str1);
```

```
Everything Is Good In Its Season
```

二、正则表达式

正则表达式就是用于字符串的匹配规则。数据类型是对象也是引用类型。正则常用于表单验证。

```
// 案例输入一个电话号码 000-88888888
var call = prompt("请输入一个电话号码");
var reg = /^\d{3}-\d{8}$/;
if(reg.test(call)){
    console.log("输入正确");
}else{
    console.log("输入错误，请重新输入");
}
```

2.1 正则的概述

正则表达式（regular expression），简称 RegExp。是被用来匹配字符串中的字符组合的模式。常用来做表单验证。

创建正则表达式最简单的方法，就是使用正则字面量。字面量：/表达式/

```
// 使用正则字面量的方法创建
var reg = /\d+/   //表示右一个或者多个数字;
var reg = /\s+/   //表示只有有一个空白字符;
var reg = /abcd/  //表示有四个字符必须是 a,b,c,d 并且顺序也是固定;
```

2.1 方法

配合正则表达式使用的方法有字符串方法和正则表达式的方法。（谁的方法只能谁调动也就是字符串的方法只能是字符串调用，正则方法只能是正则调动）

split()	根据匹配字符串切割父字符串
match()	使用正则表达式与字符串相比较，返回一个包含匹配结果的数组。
search()	对正则表达式或指定字符串进行搜索，返回首次出现的匹配项的下标。
replace()	用正则表达式和字符串直接比较，然后用新的子串来替换被匹配的子串。

```
// split()方法用字符串切割父字符串
var str = "aaabbjdkdbbbbbbbbbbjskblkk";
// 正则的匹配规则
// var reg = /b+/;
```

```
var arr = str.split(/b+/);
console.log(arr);
var str1 = "a bbb     cc   d";
var arr2 = str1.split(/\s+/);
console.log(arr2);
```

```
▶["aaa", "jdkd", "jsk", "lkk"]
▶["a", "bbb", "cc", "d"]
```

```
// match()方法用于字符串或者正则的匹配输出所在位置
var str = "abobosjkfjffkl";
console.log(str.match("obo"));
```

```
["obo", index: 2, input: "abobosjkfjffkl"]
```

如果不写 g 只输出匹配条件的第一个字符串。书写 g 会输出所有匹配的字符串。

```
var str = "abo          osjko  ofjo offkl";
console.log(str.match(/o\s+o/g));  //g 表示在全局匹配
```

```
▶["o          o", "o  o", "o o"]
```

```
// search()方法返回首次匹配的下标。没有全局搜索。
var str = "aboaosjkoaaaaaofjoaoffkl";
console.log(str.search("oo"));
console.log(str.search(/oa+o/));
```

```
2
```

```
// replace()方法表示匹配的字符串替换到原来的
var str = "www.baidu.com";
// 第一个参数表示匹配的字符串
// 第二个参数表示新的字符串。
console.log(str.replace("baidu","icketang"));
console.log(str.replace(/baidu/,"icketang"));
// 去掉字符串中的空格
var str2 = "a   b   ccccc  d";
       console.log(str2.replace(/\s+/g,""));
```

```
www.icketang.com
www.icketang.com
abccccd
```

正则的方法：

exec()。方法表示匹配的字符串在父字符串中的位置。返回一个数组。即使是全局匹配也只会返回字符串第一次出现的位置。

```
// exec()字符串在父字符串中的位置
       var str = "aaabcddddjabced";
```

```
        // 全局匹配
        console.log(str.match(/abc/g));
        // 正则方法
        // var reg = /abc/;
        console.log(/abc/.exec(str));
```

```
▶ ["abc", "abc"]
▶ ["abc", index: 2, input: "aaabcddddjabced"]
```

test()方法检测字符串中是否有符合正则表达式的部分。有返回 true，没有返回 false。

```
// test()检测字符串中是否有符合正则表达式的部分
var str = "aaabccccddddd";
console.log(/abc/.test(str));
console.log(/abcd/.test(str));
```

```
true
false
```

三、正则表达式的术语和操作符

3.1 精确匹配

正则表达式：由一些普通字符和一些特殊字符（又叫元字符--metacharacters）组成。普通字符包括大小写的字母和数字，而元字符则具有特殊的含义。

javascript 中常用特殊字符有　( ) [ ] { } \ ^ $ | ? * + .

若想匹配这类字符必须用转义符号 \ 如：\(,\^,\\

我们要匹配的正则表达式里，没有特殊符号或者操作符。我们要想匹配这些常量、普通字符，我们只能去进行精确匹配，字符串里出现的字符必须在正则里直接书写。

比如：想测试字符串”abcnddjgkgk”中是否有“abc”

```
console.log(/abc/.test("abcnddjgkgk"));
// 精确匹配，就是必须具有 abc 这三个字母，并且顺序也不能颠倒。
```

3.2 预定义特殊字符

\t　/\t/　制表符

\n　/\n/　回车符

\f　/\f/　换页符

\b　/\b/　空格

```
> console.log(/\t/.test("a    bjfjgj "))
  true
  undefined
> console.log(/\n/.test("a
  bnmmmm"))
  Uncaught SyntaxError: Invalid or unexpected token
> console.log(/\n/.test(`a
  bnmmmm`))
  true
  undefined
>
```

3.3 字符集

我们之前使用的都是一个字符匹配一个字符。

我们想用一类字符匹配一个字符。这就需要用字符集。

字符集：[],将一类字符的可能性都写在中括号之内。

简单类:正则的多个字符对应一个字符，我们可以用[]把它们括起来，让[]这个整体对应一个字符[abc]

```
var str = "sanasdnbjkdscndsbnb";
//[abc]表示一类字符集的可能性，可以是 a,可以是 b，可以是 c
console.log(str.match(/s[abc]n/g));
```

```
▶ ["san", "scn", "sbn"]
```

范围类：要匹配的字符太多，我们可以利用一个范围将可能性都包含在内。[a-z]、[0-9]、[A-Z]

```
// 范围类[a-z],[A-Z],[0-9]
console.log(/[a-z]/.test("我想看看 y 你这里有没有字母"));
console.log(/[0-9]/.test("我想看看 9 你这里有没有字母"));
```

负向类:[]前面加个元字符（^）进行取反，表示匹配不能为括号里面的字符。

```
// 负向类
var str = "sanasdnbjkdscndsbnbsknkkdkksyn";
// [^abc]表示一类字符集的可能性，不可以是 a,不可以是 b，不可以是 c
console.log(str.match(/s[^abc]n/g));
```

```
▶ ["sdn", "skn", "syn"]
```

组合类:允许用中括号匹配不同类型的单个字符。

```
// 组合类，不同类型的数据类型写一起
var str = "sanas0nbjkdscnds6nbsknkkdkksYn";
// 只要是 s*n 这种字符组合就输出
console.log(str.match(/s[a-z0-9A-Z]n/g));
```

3.4 字符的边界

^ 开头。表示字符串能够到匹配到以^后面字符串开头。（千万不能写在左中括号后面。）

```
// 开头^
```

```
console.log(/^hello/.test("hello icketang"));
console.log(/^ello/.test("ello icketang"));
```

```
true
false
```

$ 结尾。表示字符串能够匹配到以$前面的字符串结尾的字符串。

```
// 结尾$
console.log(/icketang$/.test("hello icketang"));
console.log(/icketann$/.test("hello icketang"));
```

```
true
false
```

\b 单词的边界。用于查找位于单词的开头或结尾的匹配。

```
"hello icetang".match(/\b\w+\b/);
▶ ["hello"]
"hello icketang".match(/\b\w+\s+\b/);
▶ ["hello "]
"hello icketang".match(/\b\s+\w+\b/);
▶ [" icketang"]
"hello icketang".match(/\b\w+\b/g);
▶ ["hello", "icketang"]
```

\B 非单词的边界。用于查找不处在单词的开头或结尾的匹配。

```
> "hello icketang".match(/\B\w+\B/g);
◂ ▶ ["ell", "cketan"]
> "hello icketang".match(/\B\w+\B/);
◂ ▶ ["ell"]
> "hello icketang".match(/\B\s+\w+\B/);
◂ null
```

3.5 修饰符

g 表示全局匹配。能够在全局范围内匹配。

用法：g 写在正则表达式的最后

```
/表达式/g
```

```
console.log("sanjfkfjsanjjjjjsanfff".match(/san/));
//g 表示全局匹配
console.log("sanjfkfjsanjjjjjsanfff".match(/san/g));
```

```
▶ ["san", index: 0, input: "sanjfkfjsanjjjjjsanfff"]
▶ ["san", "san", "san"]
>
```

i 表示对大小写不敏感。也就是不区分大小写

```
1  console.log("sAnjfkfjsanjjjjjsAnfff".match(/san/i));
```

```
▸ ["sAn", index: 0, input: "sAnjfkfjsanjjjjjsAnfff"]
```

可以连续书写修饰符

```
1  console.log("sAnjfkfjsanjjjjjsAnfff".match(/san/ig));
```

```
▶ ["sAn", "san", "sAn"]
```

3.6 预定义类

js 提前给我们定义好的，一些特殊字符。表示一类字符，是一些特殊字符集的简写。

. [^\n\r] 表示除了换行和回车之外的任意字符。

```
> /^.+$/.test("@#$%&");
< true
> /^.+$/.test("jfkf");
< true
> /^.+$/.test("1223");
< true
> /^.+$/.test(`a
$&@`);
< false
```

\d [0-9] 表示数字字符。

```
> /^\d+$/.test("128903")
< true
> /^\d+$/.test("12c3")
< false
> /[0-9]/.test("12c3");
< true
>
```

\D [^0-9] 表示非数字字符

```
> /^\D+$/.test("123")
< false
> /^\D+$/.test("abdd*&%")
< true
```

\w [a-zA-Z_0-9] 单词字符

word

```
> /^\D+$/.test("123")
< false
> /^\D+$/.test("abdd*&%")
< true
> /^\w+$/.test("hello");
< true
> /^\w+$/.test("hel       lo");
< false
> /^\w+$/.test("hello%&*");
< false
> /^\w+$/.test("hello99_AAA");
< true
>
```

\W [^a-zA-Z_0-9] 非单词字符

```
> /^\W+$/.test("hello");
< false
> /^\W+$/.test("he        llo");
< false
> /^\W+$/.test("          &*$%^");
< true
> /^\W+$/.test("          &*$%^90");
< false
```

\s [\t\n\x0B\f\r] 空白字符

```
> /^\s+$/.test("
< true
> /^\s+$/.test("12      ")
< false
> /^\s+$/.test(``)
< false
> /^\s+$/.test(`
`)
< true
>
```

\S [^ \t\n\x0B\f\r] 非空白字符

```
> /^\S+$/.test(`
`)
< false
> /\S+/.test("djdkjgkjkJJHH788")
< true
>
```

3.7 量词

量词用法：{}

{n}　　硬性量词，表示字符串出现 0 或者 n 次

```
/ab{2}c/.test("abbc")
true
/ab{2}c/.test("abc")
false
/ab{2}c/.test("abbbbc")
false
/a[a-z]{3}c/.test("ajjjc")
true
```

{n,m}　　软性量词，表示至少出现 n，最大不能超过 m。

```
/ab{3,6}c/.test("abbbbc")
true
/ab{3,6}c/.test("abbc")
false
/ab{3,6}c/.test("abbbbbbbbbbbbbbbbbbbbbbbbbbbc")
false
```

{n,}　软性量词，表示至少出现 n 次。

```
/ab{3,}c/.test("abbbbbbbbbbbbbbbbbbbbbbbbbbbc")
true
/ab{3,}c/.test("abbc")
false
/ab{3,}c/.test("abbbc")
true
```

?　{0,1}表示出现 0 次或者 1

```
> /ab?c/.test("ac")
< true
> /ab?c/.test("abc")
< true
> /ab?c/.test("abbc")
< false
```

+　{1,}表示出现 1 次或者多次（至少出现 1 次）

```
> /ab+c/.test("abbc")
< true
> /ab+c/.test("abc")
< true
> /ab+c/.test("ac")
< false
>
```

* {0,}，表示出现 0 次或者多次。（任意次数）

```
> /ab*c/.test("ac")
< true
> /ab*c/.test("abbbc")
< true
>
```

3.8 分组

虽然量词的出现，能帮助我们处理一排密紧相连的同类型字符。但这是不够的，我们用中括号表示范围内选择，大括号表示重复次数。如果想获取重复多个字符，我们就要用小括号进行分组了。

```
> /(bye){2}/.test("hellobyebye")
< true
> /(bye){2}/.test("hellobyebyy")
< false
> /(bye)+/.test("hellobyebye")
< true
```

3.9 或操作符

正则表达式可以使用|，操作符。

```
/a|bye/.test("aye");
true
```

表示要么是 a,要么是 bye。

```
/a|bye/.test("aye");
true
/a|bye/.test("bye");
true
/(a|bye){1}/.test("a");
true
/(a|bye){1}/.test("bye");
true
/(a|bye){1}/.test("aye");
true
/(a|bye){1}/.test("byebye");
true
/(a|bye){1}/.test("bbe");
false
```

```
/([a-z]{3})|([0-9]{2})/.test("abc")
true
/([a-z]{3})|([0-9]{2})/.test("12")
true
/([a-z]{3})|([0-9]{2}){2}/.test("1212")
true
/([a-z]{3})|([0-9]{2}){2}/.test("12")
false
```

表示要么有 3 字母要么有 2 数字

3.10 分组的反向引用

反向引用标识是对正则表达式中的匹配组捕获的子字符串进行编号，通过“\编号(在正则表达式中)”，“$编号（在正则表达式外）”进行引用。从 1 开始计数。

①在正则表达式中的编号

```
> /(bye)\1/.test("byebye")
< true
> /(bye){2}/.test("byebye")
< true
> /(bye)\1/.test("bye")
< false
> /([a-z]{3})\1/.test("abcabc")
< true
> /([a-z]{3})\1/.test("xyzabc")
< false
> /([a-z]{3})\1{2}/.test("xyzxyzxyz")
< true
```

①在正则表达式外的编号$1,$2……

```
> "123*456".replace(/(\d{3})\*(\d{3})/,"$2*$1")
< "456*123"
> "123*456".replace(/(\d{3})\*(\d{3})/,function(match,$1,$2){})
< "undefined"
> "123*456".replace(/(\d{3})\*(\d{3})/,function(match,$1,$2){
    return $2 + "*" + $1;
})
< "456*123"
> "123*456".replace(/(\d{3})\*(\d{3})/,function(match,$1,$2){
    return $1 + $2;
})
< "123456"
> "123*456".replace(/(\d{3})\*(\d{3})/,function(match,$1,$2){
    return parseInt($1) + parseInt($2);
})
< "579"
```

3.11 中文

匹配中文：[\u4e00-\u9fa5]

是一个固定用法，中文只能在正则表达式里这样表示。可以匹配简体中文或者繁体中文

```
> /^[\u4e00-\u9fa5]+$/.test("学习");
< true
> /^[\u4e00-\u9fa5]+$/.test("学a习");
< false
> /^[\u4e00-\u9fa5]+$/.test("國國國國");
< true
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```